Abstract Data Types in Event-B – An Application of Generic Instantiation

David Basin[1], Andreas Fürst[1], Thai Son Hoang[1], Kunihiko Miyazaki[2], and
Naoto Sato[2]

[1] Institute of Information Security, ETH Zurich
[2] Yokohama Research Lab, Hitachi

Abstract. Integrating formal methods into industrial practice is a challenging task. Often, different kinds of expertise are required within the same development. On the one hand, there are domain engineers who have specific knowledge of the system under development. On the other hand, there are formal methods experts who have experience in rigorously specifying and reasoning about formal systems. Coordination between these groups is important for taking advantage of their expertise. In this paper, we describe our approach of using generic instantiation to facilitate this coordination. In particular, generic instantiation enables a separation of concerns between the different parties involved in developing formal systems.

1 Introduction

Event-B is a formal method for modelling safe and reliable systems. Industrial awareness of Event-B has been enhanced by recent collaboration projects (e.g., DEPLOY [4]). These projects acted as a bridge for deploying research results in various industrial contexts with considerable success. Moreover, they also highlighted several challenges in integrating formal methods into industrial development processes. In particular, questions about interactions between developers with different kinds of expertise often arise during the deployment. On the one hand, engineers have domain knowledge including how the systems should work and why they work, but often find it challenging to formalise their reasoning. On the other hand, formal method experts, which do not have inside knowledge about the specific systems, have experience in reasoning formally about systems in general.

In this paper, we propose adapting the concept of abstract data types to Event-B to enable the interaction between the domain and formal methods experts. Abstract data types allow developers to hide implementation details that are initially irrelevant to the development of a system. As a result, systems developed with abstract data types are more intuitive and easier to verify. The realisation of the abstract data types can be done via generic instantiation by Event-B experts. In particular, the choice of which (concrete) data structure to use to represent the abstract data type can be done independently of the actual

system under development. Later, generic instantiation in Event-B enables the Event-B expert to prove that the chosen data structure is a valid realisation of the abstract data type.

Generic instantiation in Event-B was introduced in [3] and further elaborated in [8]. These works show how generic instantiation works with other standard techniques in Event-B such as refinement and composition. This paper illustrates how abstract data types can be modelled and realised using generic instantiation. Similar to our work is the recently developed *Theory Plug-in* [6]. The primary usage of the Theory Plug-in is to extend the mathematical language to include new data types. A theory module also provides an encapsulation of datatypes and enables the separation of concerns between the data types and the models that make use of them. The main difference between our work and the Theory Plug-in is that data types are usually developed together with their properties within the same theory module. As a results, the data types developed using the Theory Plug-in are usually already concrete. There is no clear separation between actual representation of data types and their abstract properties. More information on related work is in Section 5.

Structure In Section 2 we will give a brief overview of Event-B and generic instantiation in Event-B. We describe our approach in Section 3. In Section 4 we demonstrate our methodology of splitting the modelling effort on an example. In Section 5 we compare our approach with other existing approaches and in Section 6 we draw conclusions.

2   Background

2.1   The Event-B Modelling Method

Event-B [1] is a modelling method for formalising and developing systems whose components can be modeled as discrete transition systems. Event-B is centered around the general notion of *events* and its semantics is based on transition systems and simulation between such systems, as described in [1]. We will not describe in detail the semantics of Event-B here. Instead we just give a brief description of Event-B models, which are important for generic instantiation.

Event-B models are organised in terms of two basic constructs: *contexts* and *machines*. Contexts specify the static part of a model whereas machines specify the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, and *theorems*. Carrier sets are similar to types. Axioms constrain carrier sets and constants, whereas theorems are additional properties derived from axioms. The role of a context is to isolate the parameters of a formal model (carrier sets and constants) and their properties, which are intended to hold for all instances.

Machines specify behavioural properties of Event-B models. Machines may contain *variables*, *invariants* (and *theorems*), and *events*. Variables v define the state of a machine and are constrained by invariants $I(v)$. Theorems are additional properties of v derivable from $I(v)$. Possible state changes are described

by events. An event evt can be represented by the term

$$\text{evt} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } G(t,v) \textbf{ then } S(t,v) \textbf{ end } ,$$

where t stands for the event's *parameters*, $G(t,v)$ is the *guard* (the conjunction of one or more predicates) and $S(t,v)$ is the *action*. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. We use the short form evt $\mathrel{\widehat{=}} \textbf{when } G(v) \textbf{ then } S(v) \textbf{ end}$ when the event does not have any parameters, and we write evt $\mathrel{\widehat{=}} \textbf{begin } S(v) \textbf{ end}$ when, in addition, the event's guard equals *true*. A dedicated event without parameters and guard is used for the *initialisation* event (usually represented as init).

A machine can *see* multiple contexts. During the development, a context *extends* one or more contexts by declaring additional carrier sets, constants, axioms or theorems. An abstract machine can be *refined* by another concrete machine. The variables of the abstract and concrete machines are related by some gluing invariants. The existing events are refined accordingly to this relationship. Moreover, new events can be added to the concrete machine. The new events must refine a special skip event, which does not change the abstract variables.

2.2 Generic Instantiation in Event-B

Generic instantiation is a technique for reusing models by giving concrete values for abstract parameters of the models. Generic instantiation for Event-B is first mentioned in [3] and is further elaborated in [8]. We summarise the approach as follows. Suppose we have an abstract development with machines $\mathbf{M_1} \dots \mathbf{M_n}$ and their corresponding contexts $\mathbf{C_1} \dots \mathbf{C_n}$ as shown in Fig. 1. The development is generic, with the carrier sets s and constants c from the contexts $\mathbf{C_1} \dots \mathbf{C_n}$ acting as its parameters. Assume that s and c are constrained by axioms $A(s,c)$.
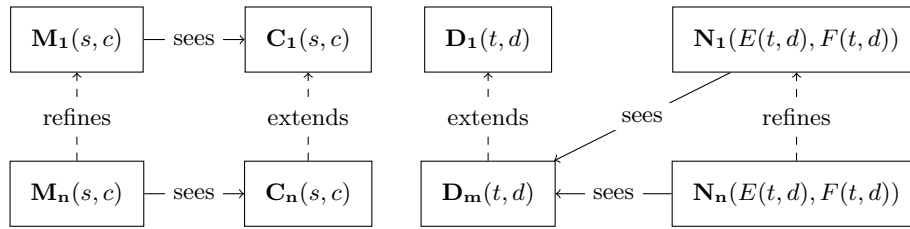

Fig. 1. Generic instantiation in Event-B

The abstract generic model can be instantiated within another development containing contexts $\mathbf{D_1} \dots \mathbf{D_m}$. Assume that the concrete contexts $\mathbf{D_1} \dots \mathbf{D_m}$ contain concrete carrier sets t and constants d, constrained by axioms $B(t,d)$. The instantiation is done by giving values for the abstract carrier sets s and

constants c in terms of concrete t and d. Let the concrete expressions $E(t,d)$ and $F(t,d)$ be the instantiated values for s and c respectively. Soundness for generic instantiation requires us to prove that the instantiated abstract axioms are derivable from the concrete axioms, i.e.,

$$B(t,d) \Rightarrow A(E(t,d), F(t,d)) \ .$$

In this paper, we further restrict the instantiation for the abstract carrier sets s so that they can only be instantiated by type-expressions, i.e. $E(t,d)$ must be some type-expressions. This is because a carrier set S in Event-B is assumed to satisfy two additional constraints (i.e., beside the stated axioms).

> non-empty: S is non-empty, i.e., $S \neq \varnothing$.
> maximal: S is maximal, i.e. $\forall x \cdot x \in S$.

The **maximal** condition is due to the fact that the Event-B models are typed. As a result, expressions used for instantiating carrier sets must be also some type-expressions, i.e., satisfying the above two conditions.

Applying generic instantiation, machines $\mathbf{N_1} \ldots \mathbf{N_n}$ are instances of $\mathbf{M_1} \ldots \mathbf{M_n}$ by syntactically replacing s and c by $E(t,d)$ and $F(t,d)$. The advantage here is that the instantiated machines are correct by construction. The resulting model can be used in conjunction with other techniques such as refinement [3] and composition [8].

3 Abstract Data Types in Event-B

An abstract data type is a mathematical model of a class of data structures. An abstract data type is typically defined in terms of the operations that may be performed on the data type with some mathematical constraints on the effects of such operations. The advantage of using an abstract data type is that the reasoning can be done purely based on the properties of the operations, regardless of the implementation. We want to use this idea in our developments. In particular the separation between the abstraction and the implementation enables us to split the work between domain experts and formal methods experts.

An abstract data type and its operations can be captured straightforwardly using contexts in Event-B. Generic instantiation can then be used to "implement" the abstract data type and prove that the actual implementation satisfies the constraints on the effects of the operations. Our approach can be summarised as follows.

Domain experts: The domain experts make use of some abstract data types and operations defined within some context to model the system in Event-B.

Formal methods experts: The formal methods experts use generic instantiation to include the details on how the abstract data types are represented and prove that the representations satisfy the assumptions of the abstract data types stated earlier.

We illustrate the use of generic instantiation by a model of the standard *stack data type*. A stack is a last in, first out (LIFO) data type that contains a collection of elements. A stack is characterised by two fundamental operations: *push* and *pop*. The *push* operation adds a new item to the top of the stack. The *pop* operation removes the stack's top element. A special constant *empty_stack* denotes the empty stack. The stack abstract data type can be modelled using a context as follows. Notice that we have defined the "type" $STACK\_TYPE$ as a carrier set and the set of possible stacks $STACK$ as a constant.

$$\textbf{sets}: STACK\_TYPE, ELEM$$

$$\textbf{constants}: STACK, empty\_stack, push, pop$$

axioms :
$\textsf{axm0\_1}: STACK \subseteq STACK\_TYPE$
$\textsf{axm0\_2}: empty\_stack \in STACK$
$\textsf{axm0\_3}: push \in STACK \times ELEM \to STACK$
$\textsf{axm0\_4}: pop \in STACK \nrightarrow STACK$
$\textsf{axm0\_5}: \mathrm{dom}(pop) = STACK \setminus \{empty\_stack\}$
$\textsf{axm0\_6}: \forall s, e \cdot s \in STACK \quad \Rightarrow \quad push(s \mapsto e) \neq empty\_stack$
$\textsf{axm0\_7}: \forall s, e \cdot s \in STACK \quad \Rightarrow \quad pop(push(s \mapsto e)) = s$

In the representation of stack data type, each stack is represented by a pair $f \mapsto n$, where f represents the content of the stack and n represents the size of the stack. Other operations of the stack data type are defined accordingly. The concrete context used for instantiation is as follows. Note that we use set comprehension to define the constants accordingly.

$$\textbf{sets}: ELEM \qquad \textbf{constants}: STACK, empty\_stack, push, pop$$

axioms :
$\textsf{axm1\_1}: STACK = \{f \mapsto n \mid n \in \mathbb{N} \wedge f \in 1 .. n \to ELEM\}$
$\textsf{axm1\_2}: empty\_stack = \varnothing \mapsto 0$
$\textsf{axm1\_3}: push = \{f, n, e \cdot$
$\qquad\qquad f \mapsto n \in STACK \wedge e \in ELEM \quad |$
$\qquad\qquad ((f \mapsto n) \mapsto e) \mapsto ((f \Leftarrow \{(n+1) \mapsto e\}) \mapsto n+1)\}$
$\textsf{axm1\_4}: pop = \{f, n \cdot f \mapsto n \in STACK \wedge n \neq 0 \quad |$
$\qquad\qquad (f \mapsto n) \mapsto ((\{n\} \vartriangleleft f) \mapsto n-1)\}$

To prove that the representation of the stack data type is consistent with the stack abstract data type, we can use instantiation where the abstract constants are instantiated with concrete constants with the same name. The abstract carrier set $STACK\_TYPE$ is instantiated with $\mathbb{P}(\mathbb{Z} \times ELEM) \times \mathbb{Z}$. The abstract axioms (i.e., $\textsf{axm0\_1} - \textsf{axm0\_7}$) must be derived from the concrete axioms (i.e., $\textsf{axm1\_1} - \textsf{axm1\_4}$). This can be done by expanding the definitions of the concrete constants accordingly.

4 Example

We illustrate our approach by modelling a set of trains on a railway network, inspired by the example in [1, Chapter 17].

4.1 Requirements Document

A railway network is divided into sections. An example of such a network is showed in Figure 2, taken from [1, Chapter 17].
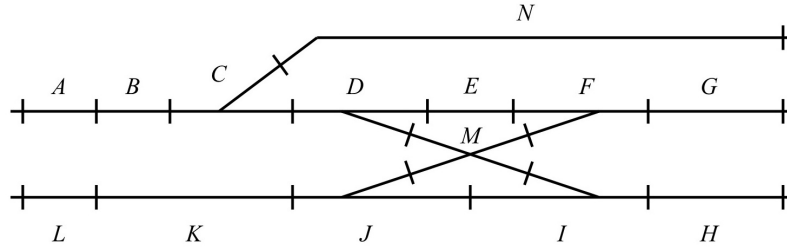

Fig. 2. Layout of a sample network with sections A to N.

A set of trains are moving within the network. Two important requirements are that trains must not derail or collide. To avoid collision, the system must ensure that each section is occupied by at most one train. Moreover, trains are assumed to move only forward within the network.

SAF 1 For each section, at most one train occupies that section.

SAF 2 Trains are always on the network.

ASM 3 Trains only move forward.

4.2 Informal Discussion

An important part of the model will formalise the trains moving within the network. Intuitively, a train can be seen as the sequence of consecutive sections that it occupies within the network. There are different possible formalisation of the trains, e.g., using functions relating occupied sections as in [1, Chapter 17], or modeling sequences as functions from integers to sections. However, the system should be correct regardless of which modelling style is used to represent the trains. In particular, the formalisation of the trains in Event-B is of little interest to the domain experts. It would be easier for the domain experts to model the trains at the more abstract level, i.e. with a train abstract data type. The decision of which representation for the train data type will be decided by the Event-B experts. In particular, different representations can be used for the train data type via separate instantiation.

4.3 Formal Model

Train Abstract Data Type We first formalise the train abstract data type in a context, focusing on requirement SAF 1. In particular, we consider the following "attributes" of a train: the sections that the train occupies (we refer to them as the train's area), the section of the train's head (the end where the train driver is sitting) and the section of the train's rear (the opposite end). This is illustrated in Figure 3.
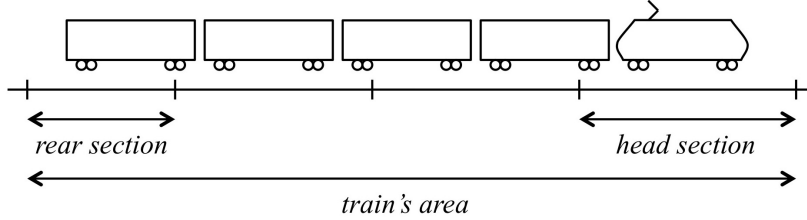
Fig. 3. Train in the network occupying sections.

Let the set of sections be a carrier set $SECTION$. We abstractly represent the trains state by a constant $TRAIN$, that is a subset of the carrier set $TRAIN\_TYPE$. Three function constants, namely $area$, $head$, and $rear$, are used to get the information about the trains' area, head position, and rear position, respectively. For an abstract data type describing a train, one can see these constants as operations of the data type.

$area$: takes a train state, returns a set of sections.
$head$: takes a train state, returns a section.
$rear$: takes a train state, returns a section.

In Event-B, we give the typing information for these constants using the following axioms.

area_Type : $area \in TRAIN \rightarrow \mathbb{P}(SECTION)$
head_Type : $head \in TRAIN \rightarrow SECTION$
rear_Type : $rear \in TRAIN \rightarrow SECTION$

Further constraints on these constants will be given later when they are needed for maintaining the correctness of the machines that use this data type.

When a train moves, the set of sections it occupies changes. When moving forward, ASM 3, the train's head reaches the end of its head section and moves to the new section ahead. Similarly, when the train's rear leaves the train's rear section, the rear is reassigned. The train's area is updated accordingly: it is extended to include the new head section when the head moves, and the rear section is removed when the rear moves. As a result, we define two additional operations for manipulating the train.

add_head: takes a train state and a section, returns a train state.
$front$:   takes a train state, returns a train state.

In Event-B, we give the type for these constant as follows.

add_head_Type : $add\_head \in TRAIN \times SECTION \nrightarrow TRAIN$
front_Type : $front \in TRAIN \nrightarrow TRAIN$

Note that we use partial functions to indicate that there are some constraints for extending the train's head and removing the train's rear.

Finally, we define an additional operation new_train to create a new train when the train enters the network from a particular section.

new_train_Type : $new\_train \in SECTION \rightarrow TRAIN$

System Model Using Train Abstract Data Type Using the train abstract data type, the system can be straightforwardly modelled. Let $TRAIN\_ID$ be the set of possible IDs for trains in the network. The variable $trains$ represents the trains currently monitored by the systems, which is a mapping from train IDs to actual trains. Initially, $trains$ is assigned the empty set $\varnothing$.

variables : $trains$
invariants :
$trains \in TRAIN\_ID \nrightarrow TRAIN$

Three events enter, extend_head, remove_rear are used to model the different cases where a train enter the network, a train extends its head to a new section, and a train removes its rear section.

enter
any t, s **where**
 $t \notin \mathrm{dom}(trains)$
 $s \in SECTION$
then
 $trains(t) := new\_train(s)$
end

extend_head
any t, s **where**
 $t \in \mathrm{dom}(trains)$
 $s \notin area(trains(t))$
then
 $trains(t) := add\_head(trains(t) \mapsto s)$
end

remove_rear
any t **where**
 $t \in \mathrm{dom}(trains)$
 $head(trains(t)) \neq rear(trains(t))$
then
 $trains(t) := front(trains(t))$
end

In particular the guard of extend_head states that the new section s is not already occupied by the train t, and the guard of remove_rear states that the head and the rear of the train t are in different sections. Moreover, these events lead us to the following constraints about the domain of operations add_head and $front$.

add_head_dom : $\mathrm{dom}(add\_head) = \{t \mapsto s \mid t \in TRAIN \wedge s \notin area(t)\}$
front_dom : $\mathrm{dom}(front) = \{t \mid t \in TRAIN \wedge head(t) \neq rear(t)\}$

An important invariant captures requirement SAF 1, stating that for any two distinct trains t_1, t_2, they do not occupy the same section.

$$\forall t_1, t_2 \cdot t_1 \in \text{dom}(trains) \wedge t_2 \in \text{dom}(trains) \wedge t_1 \neq t_2 \Rightarrow$$
$$area(trains(t_1)) \cap area(trains(t_2)) = \varnothing$$

The invariant leads to the following additional guard for enter and extend_head

$$\forall t_1 \cdot t_1 \in \text{dom}(trains) \Rightarrow s \notin area(trains(t_1))$$

While proving the correctness of our model, we discovered the following required constraints on the train abstract data type. These constraints are formalised by additional axioms over the abstract data type's operations.

area_add_head : $\forall t, s \cdot t \mapsto s \in \text{dom}(add\_head) \Rightarrow$
$$area(add\_head(t \mapsto s)) = area(t) \cup \{s\}$$
area_front : $\quad \forall t \cdot t \in \text{dom}(front) \Rightarrow area(front(t)) = area(t) \setminus \{rear(t)\}$
area_new_train : $\forall s \cdot s \in SECTION \Rightarrow area(new\_train(s)) = \{s\}$

In order to specify the fact that the trains do not derail, SAF 2, we introduce another operation, *connection*, on the train abstract data type to specify the connections of the sections belonging to a train. The typing information for *connection* is as follows.

connection_Type : $connection \in TRAIN \rightarrow (SECTION \leftrightarrow SECTION)$

The invariant corresponding to SAF 2 is

$$\forall t \cdot t \in \text{dom}(trains) \Rightarrow connection(trains(t)) \subseteq NETWORK \ ,$$

where $NETWORK$ is a constant describing the topology of the actual network. An additional guard is added to event extend_head as follows.

$$s \mapsto head(trains(t)) \in NETWORK$$

Again, we discovered additional constraints on the operation *connection* while proving the model.

connection_add_head : $\forall t, s \cdot t \mapsto s \in \text{dom}(add\_head) \Rightarrow$
$$connection(add\_head(t \mapsto s)) = connection(t) \cup \{s \mapsto head(t)\}$$
connection_front : $\quad \forall t \cdot t \in \text{dom}(front) \Rightarrow connection(front(t)) \subseteq connection(t)$
connection_new_train : $\forall s \cdot s \in SECTION \Rightarrow connection(new\_train(s)) = \varnothing$

Note that axiom connection_front does not specify exactly how a train's connection is changed when the rear is removed. It only specifies that the connection will not be enlarged. This suffices for proving the no-derailment property of the system.

Generic Instantiation We now need to find a representation for the train data type. This is the point where the role of the formal method expert becomes prominent. As mentioned before, different data structures can be used to represent the train abstract data type. We present here a solution where a train is represented by a function from an integer interval to the set of sections. Each train is associated with a tuple (a, b, f), where the interval $a \mathbin{..} b$ represents the domain of a total injective function f.

train_Def : $TRAIN = \{a \mapsto b \mapsto f \mid a \in \mathbb{Z} \wedge a \leq b \wedge f \in a \mathbin{..} b \rightarrowtail SECTION\}$

The train's head is located at the lower end of the interval (a) and its rear at the upper end (b). Injectivity guarantees that the sequence cannot include a section twice at different positions. The operations on the train data type are defined accordingly.

$$
\begin{aligned}
&\text{head\_Def :} &&head = \{a, b, f \cdot a \mapsto b \mapsto f \in TRAIN \mid (a \mapsto b \mapsto f) \mapsto f(a)\} \\
&\text{rear\_Def :} &&rear = \{a, b, f \cdot a \mapsto b \mapsto f \in TRAIN \mid (a \mapsto b \mapsto f) \mapsto f(b)\} \\
&\text{area\_Def :} &&area = \{a, b, f \cdot a \mapsto b \mapsto f \in TRAIN \mid (a \mapsto b \mapsto f) \mapsto f[a \mathbin{..} b]\} \\
&\text{add\_head\_Def :} &&add\_head = \{a, b, f, s \cdot a \mapsto b \mapsto f \in TRAIN \wedge s \notin t[a \mathbin{..} b] \\
&&&\qquad \mid (a \mapsto b \mapsto f) \mapsto s \mapsto ((a-1) \mapsto b \mapsto (f \cup \{a - 1 \mapsto s\}))\} \\
&\text{front\_Def :} &&front = \{a, b, f \cdot a \mapsto b \mapsto f \in TRAIN \wedge a \neq b \\
&&&\qquad \mid (a \mapsto b \mapsto f) \mapsto (a \mapsto (b-1) \mapsto (\{b\} \ndtriangleleft f))\} \\
&\text{new\_train\_Def :} &&new\_train = \{s \cdot s \in SECTION \mid s \mapsto (1 \mapsto 1 \mapsto \{1 \mapsto s\})\} \\
&\text{connection\_Def :} &&connection = \{a, b, f \cdot a \mapsto b \mapsto f \in TRAIN \\
&&&\qquad \mid (a \mapsto b \mapsto f) \mapsto \{i \cdot i \in a \mathbin{..} b - 1 \mid t(i) \mapsto t(i+1)\}\}
\end{aligned}
$$

By instantiating the abstract type $TRAIN\_TYPE$ to $\mathbb{Z} \times \mathbb{Z} \times \mathbb{P}(\mathbb{Z} \times SECTION)$ and other abstract constants with the concrete constants of the same name, we can prove that the constraints of the train abstract data type (abstract axioms) are derivable from the definition of the train data type.

For instantiating the train abstract data type, we used the prototype plug-in for generic instantiation.

5   Related Work

Generic instantiation in Event-B has been introduced in [3] and is further elaborated in [8]. Both papers illustrate the use of generic instantiation for reusing formal models by combining it with existing techniques like refinement and composition. In this paper, we illustrate another application of generic instantiation for algebraically modelling abstract data types. In particular, the abstract development and the concrete instantiated development enable the separation of concerns between domain experts and formal methods experts. The domain experts can work with the abstract models, stating the assumptions under which the systems work correctly. The formal method experts use generic instantiation to prove that the actual implementations satisfy the assumptions as required by the domain experts.

A similar form of generic instantiation is also available in classical B[2]. A development in classical B also contains abstract data which must be finalised when

the final software products are deployed. This finalisation process is an instantiation step, involving validating that the actual data satisfies the assumptions stated in the formal model [5]. We illustrate here (together with other work [3,8]) that generic instantiation is also useful during the stepwise development of the formal models, not just as the last realisation step in deploying the formal models.

Recent development of the *Theory Plug-in* [6] allows users to extend the mathematical languages of Event-B, e.g., by including new data types. Theorems about new data types can be stated and used later by a dedicated tactic associated with the Theory Plug-in. There is also a clear distinction between the theory modules (capturing data structures and their properties) and the Event-B models making use of the newly defined data structures. This distinction also enables a collaboration between domain experts and formal methods experts: the domain experts work with the Event-B models while the formal methods experts work with the theory modules. The difference with our approach is the order in which the work is carried out. With the Theory Plug-in, the domain experts rely on the theory developed by the formal methods experts. In our approach, the input for the formal methods experts are the abstract models that are developed by the domain experts, including the assumptions stated as axioms on the abstract carrier sets and constants. Another difference is that we can have different implementations for the abstract data types.

Our approach is similar to work on algebraic specification [7]. In this domain, a specification contains a collection of *sorts*, *operations*, and *axioms* constraining the operations. Specifications can be *enriched* by additional sorts, operations, or axioms. Furthermore, to develop programs from specifications, the specifications are transformed via a sequence of small refinement steps. During these steps, the operations are "coded" until the specification becomes a concrete description of a program. For each such refinement step, it is required to prove that the code of the operations satisfy the axioms constraining them. An algebraic specification therefore corresponds to an Event-B context, while the refinement of the algebraic specifications is similar to generic instantiation in Event-B. The main difference between algebraic specification and Event-B is that there is no corresponding elements to Event-B machines. In particular, we make use of the dynamic information of Event-B machines to derive the necessary axioms on the abstract data types.

6   Conclusion and Future Work

In this paper we presented our approach to modeling abstract data types and their implementation in Event-B. Using abstract data types allows us to hide irrelevant details that are not important for the domain expert. The domain expert can focus on modelling the functionality of the system which is his core competence. Abstract data types thereby have a similar purpose to programming interfaces in programming languages. The instantiation of the abstract data type is left to an Event-B expert. The way we introduced the concept of abstract data

types in our approach allows us to utilise generic instantiation which handles both the substitution of the abstract data type by the chosen data structure as well as the generation of the needed proof obligations to guarantee that the chosen structure is a valid instance of the abstract data type.

We successfully applied our approach to the example in this paper as well as a substantially more complex version of it. Further investigation is needed on the scalability of the approach, which is essential for its applicability in industrial development processes. Furthermore, we are interested in applying our approach outside the domain of railway systems to obtain evidence for its generality.

References

1. J-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, May 2010.
2. Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
3. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.
4. DEPLOY Project. Industrial deployment of system engineering methods providing high dependability and productivity. http://www.deploy-project.eu.
5. Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. Automated property verification for large scale b models with prob. *Formal Asp. Comput.*, 23(6):683–709, 2011.
6. Issam Maamria. Theory plug-in. http://wiki.event-b.org/index.php/Theory_Plug-in.
7. Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Asp. Comput.*, 9(3):229–269, 1997.
8. Renato Silva and Michael Butler. Supporting reuse of Event-B developments through generic instantiation. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 466–484. Springer, 2009.